Use Genetic Programming to Produce Desired Function

Haoning Jiang
University of Ottawa
75 Laurier Ave. E
Ottawa
hjian085@uottawa.ca

ABSTRACT

In this paper, I apply genetic programming methods to generate functions that are functionally equivalent to various predefined example functions. The work can generate functions that can have equivalent effects of polynomials, calculation with multiple variables, etc. The generated functions are tested with a number of test cases and are proved to be equivalent to corresponding desired functions.

Keywords

genetic programming; artificial intelligence; search based software engineering.

1. INTRODUCTION

Genetic programming has great potential of producing desired functions. Given a set of input and standard output, it is possible to use genetic programming methods to generate a function that accepts such inputs and produces corresponding standard outputs. According to genetic programming, a function program can be translated and represented as a tree whose nodes are operators and leaves are variables. These tree representations provide the possibility of applying genetic algorithm operations such as crossover and mutation on the program during evolving. By this method, it is possible to generate various possible functions that are functionally equivalent to the desired one. And some of them may have better performance on aspects like execution time or memory usage. Also, it provides a view of letting machines generate functions that we want on their own. This project is relevant to software engineering and evolutionary computing since the method used (genetic programming) is evolutionary computing based, and it solves a program optimization problem which is related to software engineering.

This report is organized as follows. Section 2 describes related work. Section 3 shows how the function programs are translated and represented as trees. This is followed by Section 4, which shows the genetic programming details of generating desired functions. Section 5 describes the results and conclusion of this project, and Section 6 presents the future work and possible improvements of this project.

2. RELATED WORK

In the paper Evolving Transformation Sequences using Genetic Algorithms, Fatiregun et al[1]. applied genetic algorithms on the optimization of source code. In their work, in order to reduce the size of source code, they regard sequences of transformations as the individuals of population in genetic algorithms and applied genetic methods such as crossover and mutation on the sequences during the evolution. Their work brings a direct and clear idea of combining genetic algorithms and software engineering.

The paper Evolutionary Improvement of Programs by David White, Andrea Arcuri and John Clark[2], inspired me to make this project. In their work, they used genetic programming to improve existing software by optimizing its non-functional properties such as execution time, memory usage, or power consumption while making sure the functional properties of the software remain unchanged. In this paper, they converted several programs into trees and applied genetic algorithm operations on it. The evolved programs passed their test cases as well as improving non-functional properties.

3. PROGRAM REPRESENTATION

3.1 Convert Programs into Trees

Genetic algorithm cannot directly applied on function programs since in most cases, if we simply apply genetic operation (for example swap several lines of source code in order to operate crossover), we may meet syntax error problems since the lines in source code have high relevance to each other and code produced via such operations may not make any since.

To solve this problem, it is necessary to convert function programs into a structure which we apply genetic operation on. Using a tree is a good idea because it is possible to apply genetic operations on it since the subtrees can be swapped in a crossover, or replaced by new random subtrees in a mutation.

To convert a function program into a tree, we first need to extract operators and variables from the function program. For example, in the function program `a + b * 3 + 16`, we have two different operators "+" and "*", and 4 variables a, b, 3, and 16. In the tree, operators are set on the node while variables are set on the leaves. The tree corresponding to the function above is shown in fig. 1.

Apart from the simple calculation operators like "+", "-" and "*", logical operators like "equal to" which returns True if two variables are equal, "larger than" which returns True if left variable is larger than right variable, and even condition operators like "if-else" can also be represented in a tree. Moreover, user customized functions can also be represented in a tree. For example, we have a function program shown in fig. 2, the corresponding tree is represented as what fig. 3 shows.

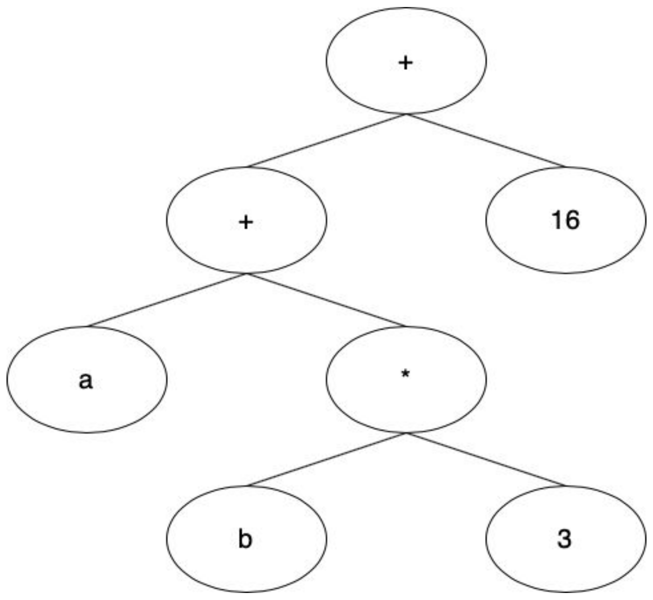
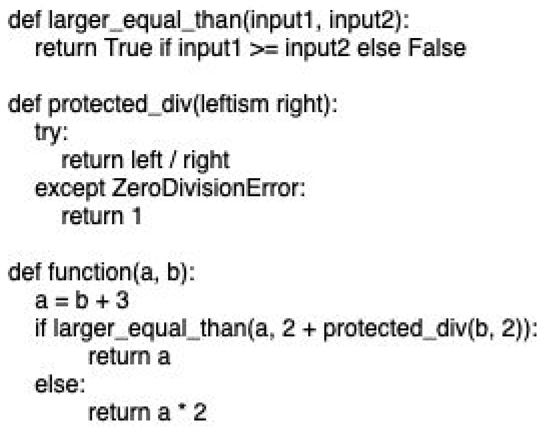


Fig. 1. The tree representation of function program a + b * 3 + 16



```
def larger_equal_than(input1, input2):
    return True if input1 >= input2 else False

def protected_div(leftism right):
    try:
        return left / right
    except ZeroDivisionError:
        return 1

def function(a, b):
    a = b + 3
    if larger_equal_than(a, 2 + protected_div(b, 2)):
            return a
    else:
            return a * 2
```

Fig. 2. Python code of a complex function program

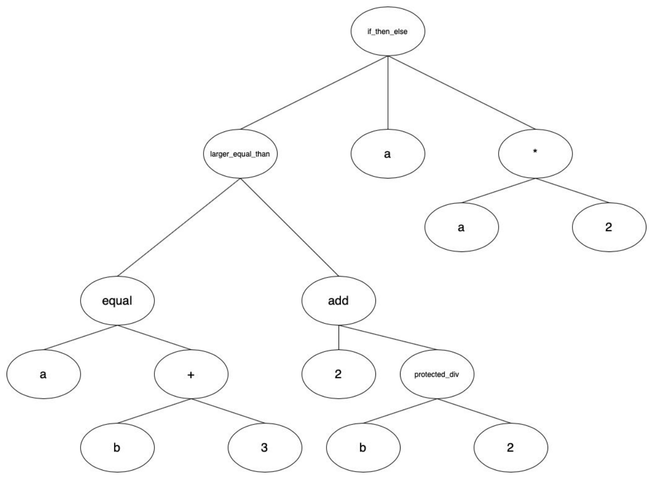


Fig. 3. Corresponding tree of the function program in Fig. 2.

3.2 Trees and Genetic Operations

As is mentioned above, we are able to apply genetic operations when using trees to represent function code. As is shown in fig. 4, we can apply crossover procedure by swapping subtrees from two trees. The mutation (shown in fig. 5) is also executable by replacing a subtree with a randomly generated tree.

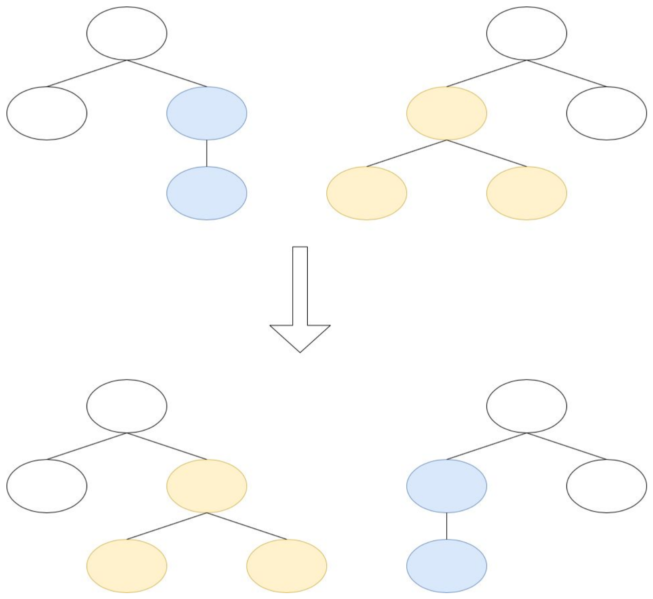


Fig. 4. Crossover example of two trees.

Besides, since we have a number of different operators and variables, we can not only translate existing function programs into trees, but also let the program generate trees by itself. The machine can combine operators and variables in a random manner and therefore randomly generate trees on its own. The fig. 5 shows some examples of trees generated by the machine itself. It is an exciting thing because once we are able to make machines randomly generate trees by itself, we are able to build an initial population in a genetic algorithm.

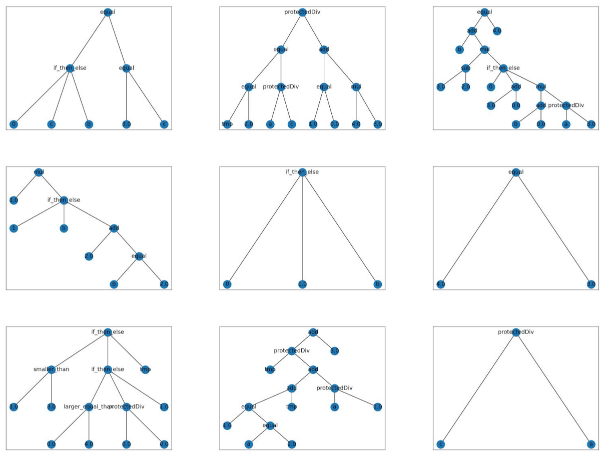


Fig. 5. Some examples of trees randomly generated by the program.

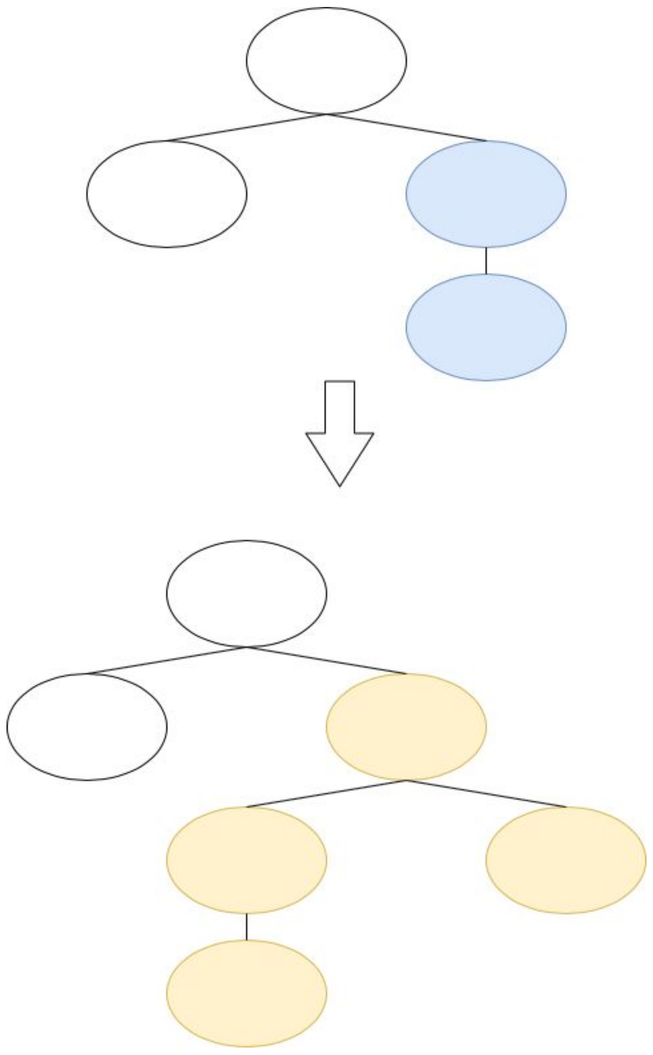


Fig. 4. Mutation example of a tree.

4. IMPLEMENTATION

4.1 Overall Implementation

The project is implemented in python, and I used Deap[3] as a tool for the genetic programming. In genetic programming, the leaves of trees are called terminals and the nodes of trees are called primitives. I used strongly typed genetic programming which defined the data type of input and output of each primitive. The limitation on the data type ensures that the machine generated trees are syntactically valid and so that the function from the trees can be executed.

The primitives defined in the genetic programming are listed in table I. Some of them are pre-defined in a function.

Table I

Primitives Used in the Project

Type	Name	Input Type	Output Type
Calculation operators	+, -, *, **, protected_div	float, float	float
Boolean	>, >=, <, <=, ==	float, float	bool
Assign	=	float, float	float
Statement	if_then_else	bool, float, float	float
Constant	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, True, False	N/A	N/A

Once set the minimum and maximum height of the tree, with the primitives listed in table I, the program is able to generate a set of trees as the population of the first generation.

The fitness varies according to different target function programs to be produced. For example, if we want to produce polynomials, the fitness can be set as the mean square difference between the result from the generated function and standard result. And the aim is to minimize this mean square difference during evolution.

The selection of next-generation individuals can be done by tournament, which is, select the individual with best fitness among a fixed number of competitors. The selected individuals are passed for the next generation's evolution.

As it mentioned in previous sections, the genetic operations like mutation and crossover are applied on the selected individuals once all individuals are selected. In the crossover procedure, two randomly selected trees are picked and their subtrees are swapped. In the mutation procedure, a randomly selected tree is picked and its subtree is replaced by a randomly generated tree.

The procedures mentioned above are repeated at each generation, and the whole evolution process is terminated after a user-determined number of generations passed.

4.1 Finding Polymorphism with Single Variable

Let's say we have a polymorphism with single variable as follow:

$$x^3 + 4x^2 + 2x + 1$$

And we want to find this polymorphism using genetic programming. The population of the first generation is set as 1000, with fitness of each individual equal to -1 since in this task, we want to minimize the fitness. In each evaluation procedure, 10 random values of x from range (-100, 100) are picked and passed to both generated function and standard function. The sum of all 10 differences between the result from the generated function and result from standard function is the fitness of this individual. The crossover rate is set as 0.5 and the mutation rate is set as 0.1, while the max number of generations is set as 400.

The program get the following result after 400 generations:

mul(mul(mul(protectedDiv(0.0, 0.0), x), add(x, 1.0)), sub(add(add(x, 1.0), add(1.0, 1.0)), protectedDiv(mul(sub(x, 1.0), add(1.0, 0.0)), mul(mul(1.0, x), add(x, 1.0)))))

The tree of this result is shown in fig. 5.

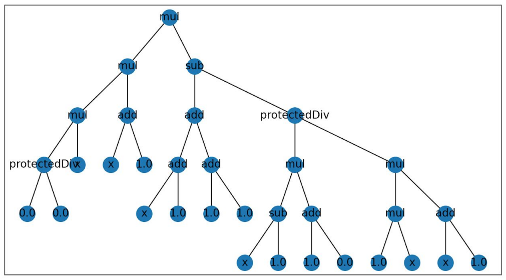


Fig. 5. The tree representation of generated function.

This function is

$$(1 * x) * x * (x + 1) * ((x + 1 + 1 + 1) - (x - 1) * 1/(1 * x * (1 + x)))$$

And it can be simplified as

$$x^3 + 4X^2 + 2x + 1$$

As we can see, it is exactly the polymorphism that we desired.

4.2 Finding Polymorphism with Multiple Variable

Let's say we have a polymorphism with 3 variable as follow:

$$a * b + b * c + a * c$$

And we want to find this polymorphism using genetic programming. The population of the first generation is set as 1000, with fitness of each individual equal to -1 since in this task, we want to minimize the fitness. In each evaluation procedure, 10 groups of random values of a, b, c from range (-100, 100) are picked and passed to both generated function and standard function. The sum of all 10 differences between the result from the generated function and result from standard function is the fitness of this individual. The crossover rate is set as 0.5 and the mutation rate is set as 0.1, while the max number of generations is set as 400.

The program get the following result after 400 generations:

add(mul(c, add(a, b)), mul(a, b))

The tree of this result is shown in fig. 6.

This function is

$$c * (a + b) + (a * b)$$

And it can be simplified as

$$a * b + b * c + a * c$$

It is equivalent to the target function we have defined at the beginning.

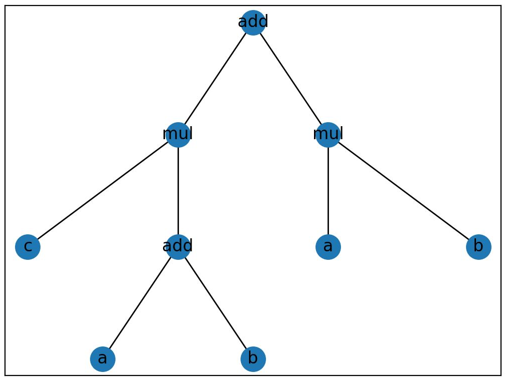


Fig. 6. The tree representation of generated function

4.2 Finding Function with Logical Operator and if-then-else Statement

Let's define a function as follow:

```
def logical_func(a, b):
  if a > b:
     return a - b
  else:
     return a + b
```

And we want to find a function that is equivalent to it generated by the program. The population of the first generation is set as 1000, with fitness of each individual equal to -1 since in this task, we want to minimize the fitness. In each evaluation procedure, 10 groups of random values of a, b from range (-100, 100) are picked and passed to both generated function and standard function. The sum of all 10 differences between the result from the generated function and result from standard function is the fitness of this individual. The crossover rate is set as 0.5 and the mutation rate is set as 0.1, while the max number of generations is set as 800.

The program get the following result after 400 generations:

add(if_then_else(larger_than(a, b), if_then_else(False, 1.0, b), sub(0.0, b)), mul(protectedDiv(a, a), add(add(mul(1.0, 0.0), mul(a, 1.0)), 0.0)))

The tree of this result is shown in fig. 7.

This function is

```
if a > b:
  if False:
     1 + (a/a * ((1 * 0 + a * 1) + 0))
  else:
     b + (a/a * ((1 * 0 + a * 1) + 0))
else:
  (0 - b) + (a/a * ((1 * 0 + a * 1) + 0))
```

This function is equivalent to

```
def logical_func(a, b):
  if a > b:
    return a - b
  else:
    return a + b
```

So, it is equivalent to the target function we have defined at the beginning.

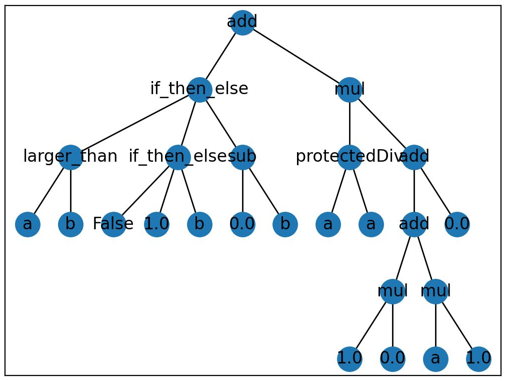


Fig. 7. The tree representation of generated function

5. RESULTS AND CONCLUSION

According to the experiments conducted in the previous section, it is proved that the program is able to generate functions as we desired.

For each case, I randomly generated a set of test cases in order to check whether the generated functions can produce the exact same result as the standard function does. And I found that the produced function of polymorphism with a single variable cannot pass all test cases and in the cases not passed, the difference of results between the generated function and standard function is usually small. I think it is due to the predefined function protected_div which returns value 1 if a value is divided by 0. Even though the simplified function is exactly the same as the standard one, the calculation using this function may have some differences.

Another finding is that the generated function always requires more execution time than standard ones. It may be due to the fact that the generated functions are not always as simple in structure as standard ones.

6. FUTURE WORK

The program is currently able to generate simple calculation and logical functions as the users desire. But when it comes to complex functions(for example functions with array or file I/O), the program may not be capable of generating an equivalent function. In the future, the program can be developed and be able to process arrays, and able to conduct switch-case operations.

7. REFERENCES

[1] D. Fatiregun, M. Harman, and R. M. Hierons, "*Evolving transformation sequences using genetic algorithms*" in Fourth IEEE International Workshop on Source Code Analysis and Manipulation, Sep. 2004, pp. 65–74, doi: 10.1109/SCAM.2004.11.

[2] D. R. White, A. Arcuri, and J. A. Clark, "*Evolutionary Improvement of Programs*" IEEE Transactions on Evolutionary Computation, vol. 15, no. 4, pp. 515–538, Aug. 2011, doi: 10.1109/TEVC.2010.2083669.

[3] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "*DEAP: a python framework for evolutionary algorithms*" in Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, Philadelphia, Pennsylvania, USA, Jul. 2012, pp. 85–92, doi: 10.1145/2330784.2330799.